Programming The Intel® Xeon Phi™ coprocessor

Alejandro Duran
Intel Software and Services Group
Legal Disclaimer

• INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPEY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

• Intel may make changes to specifications and product descriptions at any time, without notice.

• All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

• Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

• Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.

• Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

• Intel, Core, Xeon, VTune, Cilk, Intel and Intel Sponsors of Tomorrow, and Intel Sponsors of Tomorrow logo, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

• *Other names and brands may be claimed as the property of others.

• Copyright ©2011 Intel Corporation.

• Hyper-Threading Technology: Requires an Intel® HT Technology enabled system, check with your PC manufacturer. Performance will vary depending on the specific hardware and software used. Not available on all Intel® Core™ processors. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading

• Intel® 64 architecture: Requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit http://www.intel.com/info/em64t

• Intel® Turbo Boost Technology: Requires a system with Intel® Turbo Boost Technology capability. Consult your PC manufacturer. Performance varies depending on hardware, software and system configuration. For more information, visit http://www.intel.com/technology/turboboost
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture
• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming
• Real quick optimization list
• Summary
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture
• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming
• Real quick optimization list
• Summary
Timeline of Many-Core at Intel

Era of Tera CTO Keynote & "The Power Wall"

2004
Many-core technology Strategic Planning

2005
Many-core R&D agenda & BU Larrabee development

2006
Tera-scale computing research program (80+ projects)

2007
Workloads, simulators, software & insights from Intel Labs

2008
Universal Parallel Computing Research Centers

2009
1 Teraflops SGEMM on Larrabee @ SC’09

2010
Many-core applications research community

2011
Intel® Xeon Phi™ Coprocessor enters Top500 at #150 (pre-launch)
## Performance and Programmability for Highly-Parallel Processing

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core(s)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Threads</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>SIMD Width</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

**More cores**  **More Threads**  **Wider vectors**

*Product specification for launched and shipped products available on ark.intel.com.*  
1. Not launched or in planning.
Introducing Intel® Xeon Phi™ Coprocessors

Highly-parallel Processing for Unparalleled Discovery

Groundbreaking: differences

- Up to 61 IA cores/1.2GHz/ 244 Threads
- Up to 16GB memory with up to 352 GB/s bandwidth
- 512-bit SIMD instructions
- Linux operating system, IP addressable
- Standard programming languages and tools

Leading to Groundbreaking results

- Up to 1 TeraFlop/s double precision peak performance
- Enjoy up to 2.2x higher memory bandwidth than on an Intel® Xeon® processor E5 family-based server.
- Up to 4x more performance per watt than with an Intel® Xeon® processor E5 family-based server.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more information go to [http://www.intel.com/performance](http://www.intel.com/performance). Notes 1, 2 & 3, see backup for system configuration details.
Intel® Xeon Phi™ Coprocessor Product Lineup

3 Family
Outstanding Parallel Computing Solution
Performance/$ leadership

6GB GDDR5
240GB/s
>1TF DP

3120P
3120A

5 Family
Optimized for High Density Environments
Performance/watt leadership

8GB GDDR5
>300GB/s
>1TF DP
225-245W

5110P
5120D

7 Family
Highest Performance, Most Memory
Performance leadership

16GB GDDR5
352GB/s
>1.2TF DP

7120P
7120X

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance
Highly-parallel Processing for Unparalleled Discovery

*Seamlessly solve your most important problems of any scale*

---

**Intel® Xeon Phi™ product family**

- Based on Intel® Many Integrated Core (Intel® MIC) architecture
- Leading performance for highly parallel workloads
- Common Intel® Xeon® programming model seamlessly increases developer productivity
- Launching on 22nm with >50 cores

**Intel® Xeon® processor**

- Ground-breaking real-world application performance
- Industry-leading energy efficiency
- Meet HPC challenges and scale for growth

---

Single Source

Compilers and Runtimes
Parallel Performance Potential

• If your performance needs are met by an Intel Xeon® processor, they will be achieved with fewer threads than on a coprocessor.

• On a coprocessor:
  – Need more threads to achieve same performance
  – Same thread count can yield less performance

Intel Xeon Phi excels on highly parallel applications
Intel® Xeon Phi™ (Knights Corner) vs. Intel® Xeon (SNB-EP, IVB-EP)

- A companion to Xeon, not a replacement
- A ceiling lifter – KNC perspective
  - **4+x larger # of threads**
    - KNC: 60+ cores with 4 threads/core on 1 socket
  - **One package** vs. SNB-EP’s and IVB-EP’s two
  - **2x vector length** wrt Intel® Advanced Vector Extensions
    - KNC: 8 DP, 16 SP
    - SNB, IVB: 4 DP, 8 SP
  - **Higher bandwidth**
    - McCalpin Stream Triad (GB/s)
      - 175 on KNC 1.24GHz 61C, 76 on SNB 16C 2.9GHz, 101 on IVB 12C 2.7GHz
  - Instructions
    - Shorter latency on extended math instructions
Intel® Xeon Phi™ Coprocessors Workload Suitability

Can your workload scale to over 100 threads?

Yes

No

Can your workload benefit from large vectors?

Yes

No

Can your workload benefit from more memory bandwidth?

No

Yes

If application scales with threads and vectors or memory bandwidth → Intel® Xeon Phi™ Coprocessors

* Theoretical acceleration of a highly parallel processor over a Intel® Xeon® parallel processor
Intel® Xeon Phi™ Coprocessor: Increases Application Performance up to 10x

Application Performance Examples

<table>
<thead>
<tr>
<th>Customer</th>
<th>Application</th>
<th>Performance Increase[^1] vs. 2S Xeon*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Alamos</td>
<td>Molecular Dynamics</td>
<td>Up to 2.52x</td>
</tr>
<tr>
<td>Acceleware</td>
<td>8th order isotropic variable velocity</td>
<td>Up to 2.05x</td>
</tr>
<tr>
<td>Jefferson Labs</td>
<td>Lattice QCD</td>
<td>Up to 2.27x</td>
</tr>
<tr>
<td>Financial Services</td>
<td>BlackScholes SP Monte Carlo SP</td>
<td>Up to 7x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Up to 10.75x</td>
</tr>
<tr>
<td>Sinopec</td>
<td>Seismic Imaging</td>
<td>Up to 2.53x[^2]</td>
</tr>
<tr>
<td>Sandia Labs</td>
<td>miniFE (Finite Element Solver)</td>
<td>Up to 2x[^3]</td>
</tr>
<tr>
<td>Intel Labs</td>
<td>Ray Tracing (incoherent rays)</td>
<td>Up to 1.88x[^4]</td>
</tr>
</tbody>
</table>

[^1] Xeon = Intel® Xeon® processor;   
[^2] Xeon Phi = Intel® Xeon Phi™ coprocessor

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Synthetic Benchmark Summary (Intel® MKL) (5110P)

- **SGEMM (GF/s)**: Up to 2.7X, Higher is Better
- **DGEMM (GF/s)**: Up to 2.7X, Higher is Better
- **SMP Linpack (GF/s)**: Up to 2.3X, Higher is Better
- **STREAM Triad (GB/s)**: Up to 2.1X, Higher is Better

Coprocessor results: Benchmark run 100% on coprocessor, no help from Intel® Xeon® processor host (aka native)

Notes:
1. Intel® Xeon® Processor E5-2670 used for all SGEMM Matrix = 13824 x 13824, DGEMM Matrix 7936 x 7936, SMP Linpack Matrix 30720 x 30720
2. Intel® Xeon Phi™ coprocessor 5110P (ECC on) with “Gold Release Candidate” SW stack SGEMM Matrix = 11264 x 11264, DGEMM Matrix 7680 x 7680, SMP Linpack Matrix 26872 x 26872

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Source: Intel Measured results as of October 26, 2012. Configuration Details: Please reference slide speaker notes. For more information go to [http://www.intel.com/performance](http://www.intel.com/performance)
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture
• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming
• Real quick optimization list
• Summary
• Up to 61 cores
• 8GB GDDR5 Memory, 320 GB/s BW
• PCIe Gen2 (Client) x16 per direction
• ECC
MIC Architecture Overview – Features of an Individual Core

- Up to 61 in-order cores
  - Ring interconnect
- 64-bit addressing
- Two pipelines
  - Pentium® processor family-based scalar units
    - Dual issue with scalar instructions
  - Pipelined one-per-clock scalar throughput
    - 4 clock latency, hidden by round-robin scheduling of threads
- 4 hardware threads per core
  - Cannot issue back to back inst in same thread
MIC Architecture Overview – Features of an Individual Core (2)

- Intel Xeon Phi coprocessor is optimized for double precision
- All new vector unit
  - 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
    - mask registers
    - gather/scatter support
    - some transcendental functions: $\log_2, \exp_2, \frac{1}{x}, \sqrt{x}$
  - 512-bit wide vector registers per core
    - Hold 16 singles or 8 doubles per register
- Fully-coherent L1 and L2 caches
Architecture Overview
ISA/Registers

Standard Intel64 Registers (EM64T)

- rax
- rbx
- rcx
- rdx
- rsi
- rdi
- rsi
- rbp

- r8
- r9
- r10
- r11
- r12
- r13
- r14
- r15

+ 32 512bit SIMD Registers:

- zmm0
- ...
- zmm31

+ 8 mask registers (16bit wide)

- k0 (special, don’t use)
- ...
- k7

No xmm(SSE/128bit) and ymm(AVX/256bit) registers! x87 present
Intel® Xeon Phi™ Coprocessor Storage Basics

Per-core caches reference info:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Ways</th>
<th>Set conflict</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1I (instr)</td>
<td>32KB</td>
<td>4</td>
<td>8KB</td>
<td>on-core</td>
</tr>
<tr>
<td>L1D (data)</td>
<td>32KB</td>
<td>8</td>
<td>4KB</td>
<td>on-core</td>
</tr>
<tr>
<td>L2 (unified)</td>
<td>512KB</td>
<td>8</td>
<td>64KB</td>
<td>connected via core/ring interface</td>
</tr>
</tbody>
</table>

Memory:

- 8 memory controllers, each supporting 2 32-bit channels
- GDDR5 channels theoretical peak of 5.5GT/s (352 GB/s)
  - Practical peak BW between 150-180 GB/s
MIC Architecture Overview – Cache

- **L1 cache**
  - 1 cycle access
  - Up to 8 outstanding requests
  - Fully coherent

- **L2 cache**
  - 31M total across 61 cores
  - 15 cycle best access
  - Up to 32 outstanding requests
  - Fully coherent

31 MB L2

352 GB/s BW (theoretical)
More Storage Basics

Per-core TLBs reference info:

<table>
<thead>
<tr>
<th>Type</th>
<th>Entries</th>
<th>Page Size</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction</td>
<td>32</td>
<td>4KB</td>
<td>128KB</td>
</tr>
<tr>
<td>L1 Data</td>
<td>64</td>
<td>4KB</td>
<td>256KB</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>64KB</td>
<td>2MB</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2MB</td>
<td>16MB</td>
</tr>
<tr>
<td>L2</td>
<td>64</td>
<td>4KB, 64KB, or 2MB</td>
<td>Up to 128MB</td>
</tr>
</tbody>
</table>

• Note: Operating system support for 64K pages may not yet be available
Software Architecture: Two Modes

Linux* Host

- Host-side offload application
  - User code
- Offload libraries, user-level driver, user-accessible APIs and libraries
- User-level code
- System-level code

Intel® MIC Architecture support libraries, tools, and drivers

PCI-E Bus
Linux* OS

Intel® Xeon Phi™ Coprocessor

- Target-side offload application
  - User code
- Offload libraries, user-accessible APIs and libraries
- User-level code
- System-level code

Intel® MIC Architecture communication and application-launching support

PCI-E Bus
Linux* OS
Software Architecture: Two Modes

Linux* Host

- ssh or telnet connection to /dev/mic*

Intel® MIC Architecture support libraries, tools, and drivers

PCI-E Bus

Linux* OS

Intel® Xeon Phi™ Coprocessor

Target-side “native” application

User code

Standard OS libraries plus any 3rd-party or Intel libraries

Virtual terminal session

Intel® MIC Architecture communication and application-launching support

PCI-E Bus

Linux* OS
# Intel Family of Parallel Programming Models

<table>
<thead>
<tr>
<th>Intel® Cilk™ Plus</th>
<th>Intel® Threading Building Blocks</th>
<th>Domain-Specific Libraries</th>
<th>Established Standards</th>
<th>Research and Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ language extensions to simplify parallelism</td>
<td>Widely used C++ template library for parallelism</td>
<td>Intel® Integrated Performance Primitives</td>
<td>Message Passing Interface (MPI)</td>
<td>Intel® Concurrent Collections</td>
</tr>
<tr>
<td>Open sourced &amp; Also an Intel product</td>
<td></td>
<td>Intel® Math Kernel Library</td>
<td>OpenMP*</td>
<td>Offload Extensions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Coarray Fortran</td>
<td>Intel® SPMD Parallel Compiler</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OpenCL*</td>
<td></td>
</tr>
</tbody>
</table>

*Choice of high-performance parallel programming models*  

Applicable to Multicore and Many-core Programming *

* Integrated Performance Primitives not available for Intel MIC Architecture
Next Intel® Xeon Phi™ Product Family
(Codenamed Knights Landing)

- Available in Intel cutting-edge 14 nanometer process
- Stand alone CPU or PCIe coprocessor – not bound by ‘offloading’ bottlenecks
- Integrated Memory - balances compute with bandwidth

Parallel is the path forward, Intel is your roadmap!

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. Note that code name above is not the product name.
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture

• **Programming the Intel Xeon Phi coprocessor**
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming

• Real quick optimization list

• Summary
Single-source approach to Multi- and Many-Core

Develop & Parallelize Today for Maximum Performance

Use One Software Architecture Today. Scale Forward Tomorrow.
Spectrum of Programming Models and Mindsets

Multi-Core Centric
- Multi-Core Hosted
  - General purpose serial and parallel computing
- Symmetric
  - Codes with balanced needs
- Offload
  - Codes with highly-parallel phases

Many-Core Centric
- Many Core Hosted
  - Highly-parallel codes

Range of models to meet application needs
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture

• Programming the Intel Xeon Phi coprocessor
  – **Native programming**
  – Offload programming
  – Using Intel MKL
  – MPI programming

• Real quick optimization list

• Summary
How easy is to start on Intel® Xeon Phi™?

• Just add the –mmic flag to your compiler and linker (on the host)
  
  icc –o myapp.mic –mmic –O3 myapp.cpp

• Use ssh to execute
  
  ssh mic0 $(pwd)/myapp.mic
Parallelization and Vectorization are Key

- Performance increasingly depends on both threading and vectorization
- Also true for “traditional” Xeon-based computing
Wide Range of Development Options

**Threading Options**
- Intel® Math Kernel Library
- Intel® Threading Building Blocks
  - Intel® Cilk™ Plus
- OpenMP*
- Pthreads*

**Vector Options**
- Intel® Math Kernel Library
- Auto vectorization
- Semi-auto vectorization: #pragma (vector, ivdep, OpenMP)
- Array Notation: Intel® Cilk™ Plus
- C/C++ Vector Classes
  - (F32vec16, F64vec8)
- OpenCL*
- Intrinsics

Ease of use

Fine control
MIC Native Programming

• It’s all about
  – Threading
    ▪ Expose enough parallelism
    ▪ Reduce overheads
    ▪ Avoid synchronization
    ▪ Reduce load imbalance
  – Vectorizing
    ▪ Provide alias information
    ▪ Avoid strided accesses
    ▪ Avoid gathers/scatters
  – ... and memory!
    ▪ Tile your data
    ▪ Pad it correctly
    ▪ Prefetch it correctly
• Other important things
  – avoid I/O
  – use thread affinity
SIMD Types in Processors from Intel

**Intel® AVX**
- Vector size: 256bit
- Data types: 32 and 64 bit floats
- VL: 4, 8, 16
- Sample: Xi, Yi 32 bit float

**Intel® MIC**
- Vector size: 512bit
- Data types:
  - 32 and 64 bit integers
  - 32 and 64 bit floats
    (some support for 16 bits floats)
- VL: 8, 16
- Sample: 32 bit float
Auto-vectorization

• Be “lazy” and try auto-vectorization first
  – If the compiler can vectorize the code, why bother
  – If it fails, you can still deal w/ (semi-)manual vectorization

• Compiler switches of interest:
  – `vec` (automatically enabled with `-O3`)
  – `vec-report`
  – `opt-report`
Why Didn’t My Loop Vectorize?

- Linux
  - `vec-reportn`
- Windows
  - `/Qvec-reportn`

- Set diagnostic level dumped to stdout

  - `n=0`: No diagnostic information
  - `n=1`: (Default) Loops successfully vectorized
  - `n=2`: Loops not vectorized – and the reason why not
  - `n=3`: Adds dependency Information
  - `n=4`: Reports only non-vectorized loops
  - `n=5`: Reports only non-vectorized loops and adds dependency info
  - `n=6`: Much more detailed report on causes
  - `n=7`: same as 6 but not human-readable (i.e., for tools)
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_: loop was not vectorized: existence of vector dependence

35: subroutine fd( y )
36:   integer :: i
37:   real, dimension(10), intent(inout) :: y
38:   do i=2,10
39:       y(i) = y(i-1) + 1
40:   end do
41: end subroutine fd
When Vectorization Fails ...

• Most frequent reason: Data dependencies
  – Simplified: Loop iterations must be independent

• Many other potential reasons
  – Alignment
  – Function calls in loop block
  – Complex control flow / conditional branches
  – Loop not “countable”
    ▪ E.g. upper bound not a run time constant
  – Mixed data types (many cases now handled successfully)
  – Non-unit stride between elements
  – Loop body too complex (register pressure)
  – Vectorization seems inefficient
  – Many more ... but less likely to occur
Disambiguation Hints
The restrict Keyword for Pointers

Linux
- restrict
- std=c99

Windows
/Qrestrict
/Qstd=c99

- Assertion to compiler, that only the pointer or a value based on the pointer - such as (pointer+1) - will be used to access the object it points to
- Only available for C, not C++

```c
void scale(int *a, int * restrict b)
{
    for (int i=0; i<10000; i++) b[i] = z*a[i];
}

// two-dimension example:
void mult(int a[][NUM],int b[restrict][NUM]);
```
Beyond auto-vectorization

- `#pragma ivdep`, `#pragma vector`
- Cilk Plus Array notation
- OpenMP 4.0
- Vector intrinsics
OpenMP* 4.0 Specification

Released July 2013

- [http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf](http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
- A document of examples is expected to release soon

Changes from 3.1 to 4.0 (Appendix E.1):

- SIMD directives
- Device/Accelerator directives
- Taskgroup and dependant tasks
- Thread affinity
- Cancellation directives
- User-defined reductions
- Sequentially consistent atomics
- Fotran 2003 support
SIMD Support: motivation

• Provides a portable high-level mechanism to specify SIMD parallelism
  – Heavily based on Intel’s SIMD directive
• Two main new directives
  – To SIMDize loops
  – To create SIMD functions
The simd construct

#pragma omp simd [clauses]
for-loop

• where clauses can be:
  – safelen(len)
  – linear(list[:step])
  – aligned(list[:alignment])
  – private(list)
  – lastprivate(list)
  – reduction(operator:list)
  – collapse(n)

• Instructs the compiler to try to SIMDize the loop even if it cannot guarantee that is dependence free
• Loop needs to be in “Canonical form”
  – as in the loop worksharing construct
The simd construct clauses

• **safelen** (length)
  – Maximum number of iterations that can run concurrently without breaking a dependence
    ▪ in practice, maximum vector length - 1

• **linear** (list[:linear-step])
  – The variable value is in relationship with the iteration number
    ▪ \( x_i = x_\text{orig} + i \times \text{linear-step} \)

• **aligned** (list[:alignment])
  – Specifies that the list items have a given alignment
  – Default is alignment for the architecture
The simd construct

```c
#pragma omp parallel for schedule(guided)
    for (int32_t y = 0; y < ImageHeight; ++y) {
        double c_im = max_imag - y * imag_factor;
        fcomplex v = (min_real) + (c_im * 1.0iF);
        #pragma omp simd linear(v:real_factor)
            for (int32_t x = 0; x < ImageWidth; ++x) {
                count[y][x] = mandel(v, max_iter);
                v += real_factor;
            }
    }
```

Function call might result in inefficient vectorization
#pragma omp simd safelen(4)
for ( int i = 0; i < n; i += 20 )
    a[i] = a[i-100] * b[i];

maximum distance between safe iterations is 4

a[5] and a[0] have a dependence

Up to 5 iterations could run concurrently (a[0]-a[4])
The declare simd construct

```c
#pragma declare simd [clauses]
#pragma declare simd [clauses]]
```

function definition or declaration

- where clauses can be:
  - `simdlen`(length)
  - `uniform`(argument-list)
  - `inbranch`
  - `notinbranch`
  - `linear`(argument-list[:step])
  - `aligned`(argument-list[:alignment])
  - `reduction`(operator:list)

- Instructs the compiler to generate SIMD enable version(s) that can be used from SIMD loops
The declare simd construct

```c
#pragma omp declare simd uniform(max_iter)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
        z = z * z + c;
        int t = (cabsf(z) < 2.0f);
        count += t;
        if (t == 0) { break;}
    }

    return count;
}
```

Now the previous loop will use a SIMD-enabled version of the mandel function
The declare simd clauses

- **simdlen**(length)
  - generate function to support a given vector length

- **uniform**(argument-list)
  - argument has a constant value between the iterations of a given loop

- **inbranch**
  - function always called from inside an if statement

- **notinbranch**
  - function never called from inside an if statement
SIMD combined constructs

• Worksharing + SIMD

  #pragma omp for simd [clauses]

  – First distribute the iterations among threads, then vectorize the resulting iterations

• Parallel + worksharing + SIMD

  #pragma omp parallel for simd [clauses]
OpenMP Affinity

• OpenMP currently only supports OpenMP Thread to HW thread affinity
  – memory affinity is “supported” implicitly
• The **OMP_PROC_BIND** environment variable controls how threads are mapped to HW threads
  – **false**, means threads are not bound (OS maps them)
  – **true**, means bind the threads
    ▪ but not how
  – list of **master, close, spread**
    ▪ imply true

Intel OpenMP Runtime had KMP_AFFINITY environment variable for a long time
OpenMP Places

• Controlled via the **OMP_PLACES** environment variable
  – List of processor sets
    ▪ **OMP_PLACES**="\{0,1,2,3\},\{4,5,6,7\}"
    ▪ **OMP_PLACES**="\{0:4:2\},\{1:4:2\}"
  – Abstract names
    ▪ **OMP_PLACES**="cores(8)"
Affinity and Places

• Policies / affinity types:
  – **Master**: keep worker threads in the same place partition as the master thread
  – **Close**: keep worker threads “close” to the master thread in contiguous place partitions
  – **Spread**: create a sparse distribution of worker threads across the place partitions

• The parallel construct also has a `proc_bind` clause
  – only has effect if `proc-bind` is not false
Examples: master

- OMP_PLACES="cores(8)"

master 2

master 4

master 8
Example: close

- For data locality, load-balancing, and more dedicated-resources
  - select OpenMP threads near the place of the master
  - wrap around once each place has received one OpenMP thread

```
close 2

master worker partition

p0 p1 p2 p3 p4 p5 p6 p7
```

```
close 4

master worker partition

p0 p1 p2 p3 p4 p5 p6 p7
```

```
close 8

master worker partition

p0 p1 p2 p3 p4 p5 p6 p7
```
Example: spread

• For load balancing, most dedicated hardware resources
  – spread OpenMP threads as evenly as possible among places
  – create sub-partition of the place list
    ▪ subsequent threads will only be allocated within sub-partition

spread 2

spread 8

spread 16
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture
• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming
• Real quick optimization list
• Summary
Device Model

- OpenMP 4.0 supports accelerators and coprocessors
- Device model:
  - One host
  - Multiple accelerators/coprocessors of the same kind
The target construct

`#pragma omp target [clauses]`

*structured block*

• where clauses can be:
  – **device** (*num-device*)
  – **map** ( [alloc | to | from | tofrom : ] list )
  – if (*expr*)
The target construct

• The associated structured block will be synchronously executed in an attached device
  – device specified by the `device` clause or the `default-device-var` ICV
    ▪ set by `OMP_DEFAULT_DEVICE` or `omp_set_default_device()`
• Execution starts in the device with one thread
int a[N], int res;

#pragma omp target
{
    for (int i = 0; i < N; i++)
        res += a[i];
}

printf("result = %d\n", res);

Runs on the device with one thread!
Target construct example

```c
int a[N], int res;
#pragma omp target
{
    #pragma omp parallel for reduction(+:res)
    for (int i = 0; i < N; i++)
        res += a[i];
}
printf("result = %d \n", res);
```

Runs on the device with N threads

By default data is copied in & out of the device
The map clause

• Specifies how data is moved between the host and the device
  – **to**
    ▪ copy on entry to the region from host do device
  – **from**
    ▪ copy on exit of the region from device to host
  – **tofrom**
    ▪ default specifier
  – **alloc**
    ▪ creates a private copy in the device that is not synchronized

• Storage is reused if already existed on the device
Target construct example

```c
int a[N], int res;

#pragma omp target map(to:a) map(from:res)
{
    #pragma omp parallel for reduction(+:res)
    for (int i = 0; i < N; i++)
        res += a[i];
}

printf("result = %d\n", res);
```
The target data construct

#pragma omp target data [clauses]
  structured block

- where clauses are:
  - device (num-device)
  - map ( alloc | to | from | tofrom : list )
  - if (expr)
The target data construct

- Data is moved between the host and the device but execution is **NOT** transferred to the device
- Allows data to persist across multiple **target** regions
#pragma omp target data device(0) map(alloca:tmp[0:N]) map(to:input[:N])
    map(from:result)
{
    #pragma omp target device(0)
    #pragma omp parallel for
        for (i=0; i<N; i++)
            tmp[i] = some_computation(input[i], i);

do_some_other_stuff_on_host();

    #pragma omp target device(0)
    #pragma omp parallel for reduction(+:result)
        for (i=0; i<N; i++)
            result += final_computation(tmp[i], i)
}
The target update construct

`#pragma omp target update [clauses]`

- where clauses are
  - `to (list)`
  - `from (list)`
  - `device (num-device)`
  - `if (expression)`

- Allows to update the value from/to the device in the middle of `target data` region
#pragma omp target data device(0) map(aloc:tmp[0:N]) map(to:input[:N])
    map(from:result)
{
    #pragma omp target device(0)
    #pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    get_new_input_from_neighbour();
    #pragma target update device(0) to(input[:N])

    #pragma omp target device(0)
    #pragma omp parallel for reduction(+:result)
    for (i=0; i<N; i++)
        result += final_computation(tmp[i], i)
}
The declare target construct

• C/C++

#pragma omp declare target
declarations-or-definitions
#pragma omp end declare target

• Fortran

!$omp declare target(list)

• Allows to declare variables and functions that will be used from a target region
Declare target construct example

```c
#pragma omp declare target
int a[N];

int foo ( int i )
{
    return a[i];
}
#pragma omp end declare target

int res;
#pragma omp target map(from: res)
{
    #pragma omp parallel for reduction(+: res)
    for ( int i = 0; i < N; i++ )
        res += foo(i);
}
printf("result = %d\n",res);
```
Asynchronous Offload

- OpenMP accelerator constructs rely on existing OpenMP features to implement asynchronous offloads.

```c
#pragma omp parallel
#pragma omp single
{
#pragma omp task
{
#pragma omp target map(in:input[:N]) map(out:result[:N])
#pragma omp parallel for
  for (i=0; i<N; i++) {
    result[i] = some_computation(input[i], i);
  }
}

do_something_important_on_host();

#pragma omp taskwait
}
```
team Constructs

#pragma omp team [clauses]

structured-block

Clauses: num_teams( integer-expression )
num_threads( integer-expression )
default(shared | none)
private( list )
firstprivate( list )
shared( list )
reduction( operator : list )

• If specified, a teams construct must be contained within a target construct. That target construct must contain no statements or directives outside of the teams construct.

• distribute, parallel, parallel loop, parallel sections, and parallel workshare are the only OpenMP constructs that can be closely nested in the teams region.
Distribute Constructs

```c
#pragma omp distribute [clauses]
  for-loops
  Clauses: private( list )
            firstprivate( list )
            collapse( n )
            dist_schedule( kind[, chunk_size] )
```

- A `distribute` construct must be closely nested in a `teams` region.
#pragma omp target device(0)
#pragma omp teams num_teams(60) num_threads(4) // 60 physical cores, 4 h/w threads each
{
#pragma omp distribute //this loop is distributed across teams
    for (int i = 0; i < 2048; i++) {
#pragma omp parallel for // loop is executed in parallel by all threads (4) of the team
    for (int j = 0; j < 512; j++) {
#pragma omp simd // create SIMD vectors for the machine
        for (int k=0; k<32; k++) {
            foo(i,j,k);
        }
    }
}
}
Options for Offloading Application Code

• Intel Composer XE 2011 for MIC supports two models:
  – Offload pragmas
    ▪ Only trigger offload when a MIC device is present
    ▪ **Safely ignored by non-MIC compilers**
  – Offload keywords
    ▪ Only trigger offload when a MIC device is present
    ▪ **Language extensions, need conditional compilation to be ignored**

• Offloading and parallelism is orthogonal
  – Offloading only transfers control to the MIC devices
  – Parallelism needs to be exploited by a second model (e.g. OpenMP*)
Heterogeneous Compiler
Data Transfer Overview

- The host CPU and the Intel Xeon Phi coprocessor do not share physical or virtual memory in hardware.

Two offload data transfer models are available:

1. Explicit Copy
   - Programmer designates variables that need to be copied between host and card in the offload directive.
   - Syntax: Pragma/directive-based.
   - C/C++ Example: #pragma offload target(mic) in(data:length(size))
   - Fortran Example: !dir$ offload target(mic) in(a1:length(size))

2. Implicit Copy
   - Programmer marks variables that need to be shared between host and card.
   - The same variable can then be used in both host and coprocessor code.
   - Runtime automatically maintains coherence at the beginning and end of offload statements.
   - Syntax: keyword extensions based.
   - Example: _Cilk_shared double foo; _Offload func(y);
float reduction(float *data, int numberOf)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(numberOf))
    {
        #pragma omp parallel for reduction(+:ret)
            for (int i=0; i < numberOf; ++i)
                ret += data[i];
    }
    return ret;
}

Note: copies numberOf elements to the coprocessor, not
numberOf*sizeof(float) bytes – the compiler knows data’s type
Heterogeneous Compiler Offload using Implicit Copies

- Section of memory maintained at the same virtual address on both the host and Intel® MIC Architecture coprocessor
- Reserving same address range on both devices allows
  - Seamless sharing of complex pointer-containing data structures
  - Elimination of user marshaling and data management
  - Use of simple language extensions to C/C++

![Diagram showing the relationship between Host Memory, C/C++ executable, Offload code, and Intel® MIC Memory with the same address range.](image)
Heterogeneous Compiler
Implicit: Offloading using _Offload Example

// Shared variable declaration for pi
_Cilk_shared float pi;

// Shared function declaration for compute
// compute
_Shared void compute_pi(int count)
{
    int i;

    #pragma omp parallel for \
        reduction(+:pi)
    for (i=0; i<count; i++)
    {
        float t = (float)((i+0.5f)/count);
        pi += 4.0f/(1.0f+t*t);
    }
}

_Offload compute_pi(count);
LEO advantages over OpenMP 4.0

• Implicit offloading support
• Unstructured memory management
• Asynchronous data transfers
• Asynchronous offload regions
• Offload regions dependencies
**LEO: signals and memory control**

This does nothing except allocating an array

```c
#pragma offload_transfer target(mic:0) \
  nocopy(in1:length(cnt)) alloc_if(1) free_if(0))
```

Start an asynchronous transfer, tracking signal in1

```c
#pragma offload_transfer target(mic:0) \
  in(in1:length(cnt) alloc_if(0) free_if(0)) signal(in1)
```

Start once the completion of the transfer of in1 in signaled

```c
#pragma offload target(mic:0) nocopy(in1) wait(in1) \
  out(res1:length(cnt) alloc_if(0) free_if(0))
```

This does nothing except freeing an array

```c
#pragma offload_transfer target(mic:0) \
  nocopy(in1:length(cnt) alloc_if(0) free_if(1))
```
Asynchronous Transfer & Double Buffering

- Overlap computation and communication
- Generalizes to data domain decomposition

```
Host               Target

pre-work

iteration 0

iteration 1

iteration n

iteration n+1

last iteration
```
Double Buffering I

```c
int main(int argc, char* argv[]) {
  // ... Allocate & initialize in1, res1,
  //... in2, res2 on host
  #pragma offload_transfer target(mic:0) in(cnt)\
  nocopy(in1, res1, in2, res2 : length(cnt) \
  alloc_if(1) free_if(0))

  do_async_in();

  #pragma offload_transfer target(mic:0) \ 
  nocopy(in1, res1, in2, res2 : length(cnt) \ 
  alloc_if(0) free_if(1))
  return 0;
}
```

Only allocate arrays on card with alloc_if(1), no transfer

Only free arrays on card with free_if(1), no transfer
void do_async_in() {
    float lsum;
    int i;
    lsum = 0.0f;
    
    #pragma offload_transfer target(mic:0) in(in1 : length(cnt) \ 
    alloc_if(0) free_if(0)) signal(in1)
    for (i = 0; i < iter; i++) {
        if (i % 2 == 0) {
            #pragma offload_transfer target(mic:0) if(i != iter - 1) \ 
            in(in2 : length(cnt) alloc_if(0) free_if(0) alloc_if(0) free_if(0)) signal(in2)
        }
    }
    #pragma offload target(mic:0) nocopy(in1) wait(in1) \ 
    out(res1 : length(cnt) alloc_if(0) free_if(0))
    {
        compute(in1, res1);
    }
    lsum = lsum + sum_array(res1);
} else {
    ...

Send buffer in1

Send buffer in2

Once in1 is ready (signal!) process in1
Double Buffering III

...} else {

#pragma offload_transfer target(mic:0) if(i != iter - 1) \ in(in1 : length(cnt) alloc_if(0) free_if(0)) signal(in1)

#pragma offload target(mic:0) nocopy(in2) wait(in2) \ out(res2 : length(cnt) alloc_if(0) free_if(0))

  { compute(in2, res2);
  }
  lsum = lsum + sum_array(res2);

async_in_sum = lsum / (float)iter;
  } // for
} // do_async_in()
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture
• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming
• Real quick optimization list
• Summary
What is MKL?

Intel® MKL is industry’s leading math library *

- Linear Algebra
  - BLAS
  - LAPACK
  - Sparse solvers
  - ScaLAPACK

- Fast Fourier Transforms
  - Multidimensional (up to 7D)
  - FFTW interfaces
  - Cluster FFT

- Vector Math
  - Trigonometric
  - Hyperbolic
  - Exponential, Logarithmic
  - Power, Root
  - Rounding

- Vector Random Number Generators
  - Congruential
  - Uniform
  - Recursive
  - Wichmann-Hill
  - Mersenne Twister
  - Sobol
  - Neiderreiter
  - Non-deterministic

- Summary Statistics
  - Kurtosis
  - Variation coefficient
  - Quantiles, order statistics
  - Min/max
  - Variance-covariance
  - ...

- Data Fitting
  - Splines
  - Interpolation
  - Cell search

* 2011 & 2012 Evans Data N. American developer surveys

Source

Multicore CPU

Multicore CPU

Intel® Xeon Phi™ coprocessor

Clustering

Clusters with Multicore and Many-core

Multicore

Many-core

Clusters
Intel® MKL Supports for Intel MIC

• Intel® MKL 11.0 beta supports the Intel® Xeon Phi™ coprocessor
• Heterogeneous computing
  ▪ Takes advantage of both multicore host and many-core coprocessors
• Optimized for wider (512-bit) SIMD instructions
• Flexible usage models:
  ▪ Automatic Offload: Offers transparent heterogeneous computing
  ▪ Compiler Assisted Offload: Allows fine offloading control
  ▪ Native execution: Use MIC coprocessors as independent nodes

Using Intel® MKL on Intel MIC architecture
• Performance scales from multicore to many-cores
• Familiarity of architecture and programming models
• Code re-use, Faster time-to-market
Intel® MKL Usage Models on Intel MIC

• Automatic Offload
  - No code changes required
  - Automatically uses both host and target
  - Transparent data transfer and execution management

• Compiler Assisted Offload
  - Explicit controls of data transfer and remote execution using compiler offload pragmas/directives
  - Can be used together with Automatic Offload

• Native Execution
  - Uses MIC coprocessors as independent nodes
  - Input data is copied to targets in advance
Automatic Offload (AO)

- Offloading is automatic and transparent
- By default, Intel® MKL decides:
  - When to offload
  - Work division between host and targets
- Users enjoy host and target parallelism automatically
- Users can still control work division to fine tune performance
How to Use Automatic Offload

• Using Automatic Offload is easy

Call a function:

mkl_mic_enable()

or

Set an env variable:

MKL_MIC_ENABLE=1

• What if there doesn’t exist a MIC card in the system?
  ▪ Runs on the host as usual **without any penalty!**
Automatic Offload Enabled Functions

- A selective set of MKL functions are subject to AO
  - Only functions with sufficient computation to offset data transfer overhead are subject to AO

- In 11.0.2, only these functions are AO enabled:
  - Level-3 BLAS: ?GEMM, ?TRSM, ?TRMM
  - LAPACK: LU (?GETRF), Cholesky ((S/D)POTRF), and QR (?GEQRF) factorization functions
  - plus functions using the above ones!
  - AO support will be expanded in future updates.
Work Division Control in Automatic Offload

- Using support functions

<table>
<thead>
<tr>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_MIC_Set_Workdivision(MKL_TARGET_MIC, 0, 0.5)</td>
<td>Offload 50% of computation only to the 1st card.</td>
</tr>
</tbody>
</table>

- Using environment variables

<table>
<thead>
<tr>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_MIC0_WORKDIVISION=50</td>
<td>Offload 50% of computation only to the 1st card.</td>
</tr>
</tbody>
</table>

- The support functions take precedence over environment variables
Compiler Assisted Offload (CAO)

- Offloading is explicitly controlled by compiler pragmas or directives
- All MKL functions can be offloaded in CAO
  - In comparison, only a subset of MKL is subject to AO
- Can leverage the full potential of compiler’s offloading facility
- More flexibility in data transfer and remote execution management
  - A big advantage is data persistence: Reusing transferred data for multiple operations
How to Use Compiler Assisted Offload

• The same way you would offload any function call to MIC

• An example in C:

```c
#pragma offload target(mic) \
   in(transa, transb, N, alpha, beta) \
   in(A:length(matrix_elements)) \
   in(B:length(matrix_elements)) \
   in(C:length(matrix_elements)) \
   out(C:length(matrix_elements) alloc_if(0))
{
   sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, 
         &beta, C, &N);
}
```
Suggestions on Choosing Usage Models

• Choose native execution if
  ▪ Highly parallel code
  ▪ Want to use MIC cards as independent compute nodes, and
  ▪ Use only MKL functions that are optimized for MIC (see “Performance on KNC” slides)

• Choose AO when
  ▪ A sufficient Byte/FLOP ratio makes offload beneficial
  ▪ BLAS level 3 functions
  ▪ LU, Cholesky, and QR factorization

• Choose CAO when either
  ▪ There is enough computation to offset data transfer overhead
  ▪ Transferred data can be reused by multiple operations

• You can always run on the host if offloading does not achieve better performance
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture

• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
    – **MPI programming**

• Real quick optimization list

• Summary
Spectrum of Programming Models and Mindsets

Multi-Core Centric
- Xeon
  - Multi-Core Hosted
    - General purpose serial and parallel computing
  - Symmetric
    - Codes with balanced needs
- Offload
  - Codes with highly-parallel phases
  - Multi-core (Xeon)
  - Many-core (MIC)
  - Main()
  - Foo()
  - MPI_*()
  - Main()
  - Foo()
  - MPI_*()
  - Main()
  - Foo()
  - MPI_*()
  - Main()
  - Foo()
  - MPI_*()
  - Main()
  - Foo()
  - MPI_*()

Many-Core Centric
- MIC
  - Many Core Hosted
    - Highly-parallel codes
Levels of communication speed

- Current clusters are not homogenous regarding communication speed:
  - Inter node (Infiniband, Ethernet, etc)
  - Intra node
    ▪ Inter sockets (Quick Path Interconnect)
    ▪ Intra socket

- Two additional levels to come with MIC co-processor:
  - Host-MIC communication
  - Inter MIC communication
Selecting network fabrics

- Intel® MPI selects automatically the best available network fabric it can find.
  - Use I_MPI_FABRICS to select a different communication device explicitly
- The best fabric is usually based on Infiniband (dapl, ofa) for inter node communication and shared memory for intra node
- Available for KNC:
  - shm, tcp, ofa, dapl
  - Availability checked in the order shm:dapl, shm:ofa, shm:tcp (intra:inter)
- Set I_MPI_SSHM_SCIF=1 to enable shm fabric between host and MIC
Co-processor only Programming Model

- MPI ranks on Intel® MIC (only)
- All messages into/out of Intel® MIC coprocessors
- Intel® Cilk™ Plus, OpenMP®, Intel® Threading Building Blocks, Pthreads used directly within MPI processes

**Build Intel® MIC binary using Intel® MIC compiler.**

**Upload the binary to the Intel® MIC Architecture.**

**Run instances of the MPI application on Intel® MIC nodes.**
**Co-processor-only Programming Model**

- MPI ranks on the MIC coprocessor(s) only
- MPI messages into/out of the MIC coprocessor(s)
- Threading possible
  - Build the application for the MIC Architecture
    ```bash
    # mpiicc -mmic -o test_hello.MIC test.c
    ```
  - Upload the MIC executable
    ```bash
    # scp ./test_hello.MIC mic0:/tmp/
    Remark: If NFS available no explicit uploads required (just copies)!
    ```
  - Launch the application on the co-processor from host
    ```bash
    # I_MPI_MIC=enable mpirun -n 2 -wdir /tmp -host mic0 ./test_hello.MIC
    ```
  - Alternatively: login to MIC and execute the already uploaded mpirun there!
Symmetric Programming Model

- MPI ranks on Intel® MIC Architecture and host CPUs
- Messages to/from any core
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* used directly within MPI processes

Build Intel® 64 and Intel® MIC Architecture binaries by using the resp. compilers targeting Intel® 64 and Intel® MIC Architecture.

Upload the Intel® MIC binary to the Intel® MIC Architecture.

Run instances of the MPI application on different mixed nodes.
Symmetric model

- MPI ranks on the MIC coprocessor(s) and host CPU(s)
- MPI messages into/out of the MIC(s) and host CPU(s)
- Threading possible
  - Build the application for Intel®64 and the MIC Architecture separately
    # mpiicc -o test_hello test.c
    # mpiicc -mmic -o test_hello.MIC test.c
  - Upload the MIC executable
    # scp ./test_hello.MIC mic0:/tmp/
  - Launch the application on the host and the co-processor from the host
    # export I_MPI_MIC=enable
    # mpirun -n 2 -host <hostname> ./test_hello : -wdir /tmp -n 2 -host mic0 ./test_hello.MIC
MPI+Offload Programming Model

- MPI ranks on Intel® Xeon® processors (only)
- All messages into/out of host CPUs
- Offload models used to accelerate MPI ranks
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* within Intel® MIC

Build Intel® 64 executable with included offload by using the Intel® 64 compiler.

Run instances of the MPI application on the host, offloading code onto MIC.

Advantages of more cores and wider SIMD for certain applications
MPI+Offload Programming Model

- MPI ranks on the host CPUs only
- MPI messages into/out of the host CPUs
- Intel® MIC Architecture as an accelerator
  - Compile for MPI and internal offload
    # mpiicc -o test test.c
  - Compiler compiles by default for offloading if offload construct is detected!
    - Switch off by using the -no-offload flag
  - Execute on host(s) as usual
    # mpiexec -n 2 ./test
  - MPI processes will offload code for acceleration
Hybrid Computing

• Combine MPI programming model with threading model
• Overcome MPI limitations by adding threading:
  – Potential memory gains in threaded code
  – Better scalability (e.g. less MPI communication)
  – Threading offers smart load balancing strategies
• Result: Maximize performance by exploitation of hardware (incl. co-processors)
Intel® MPI Support of Hybrid Codes

• Intel® MPI is strong in mapping control
• Sophisticated default or user controlled
  - `I_MPI_PIN_PROCESSOR_LIST` for pure MPI
  - For hybrid codes (takes precedence):
    ```
    I_MPI_PIN_DOMAIN = <size>[::<layout>]
    }
    <size> =
    omp Adjust to OMP_NUM_THREADS
    auto #CPUs/#MPIprocs
    <n> Number
    }
    <layout> =
    platform According to BIOS numbering
    compact Close to each other
    scatter Far away from each other
    ```
• Naturally extends to hybrid codes on MIC
Intel® MPI Support of Hybrid Codes

- Define `I_MPI_PIN_DOMAIN` to split logical processors into non-overlapping subsets
- Mapping rule: 1 MPI process per 1 domain

Pin OpenMP threads inside the domain with `KMP_AFFINITY` (or in the code)
The execution command `mpiexec` of Intel® MPI reads argument sets from the command line:

- Sections between `"::"` define an argument set (also lines in a configfile, but not yet available in Beta)
- Host, number of nodes, but also environment can be set independently in each argument set

```
# mpiexec -env I_MPI_PIN_DOMAIN 4 -host myXeon ...
   : -env I_MPI_PIN_DOMAIN 16 -host myMIC
```

Adapt the important environment variables to the architecture

- `OMP_NUM_THREADS, KMP_AFFINITY` for OpenMP
- `CILK_NWORKERS` for Intel® Cilk™ Plus
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture

• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming

• **Real quick optimization list**

• Summary
OpenMP performance

- Extract as much parallelism as possible
  - Use collapse clause
  - Consider “replication”
- Avoid load imbalance
- Avoid sequential code
  - Avoid locking
  - Avoid atomic operations
- Fuse regions if possible
  - both parallel and worksharing regions
- Use thread affinity
  - I_MPI_PIN_DOMAIN, KMP_AFFINITY
  - avoid OS core
- Use as coarse parallelism as possible
- Tune number of threads per core
  - try optional -opt-threads-per-core=n
SIMD performance checklist

- Check vector report
- Remove aliasing
- Use simd directives when combined with OpenMP
- Ensure alignment of data
- Avoid gather/scatters
  - Use stride 1 when possible
  - Use SoA instead of AoS
- Consider peeling loops to avoid boundary conditions
- Use signed 32bit integer
- Use single precision when possible
  - control precision
- Use “x*1/const” instead of “x/const”
- Consider padding to avoid reminders
  - -opt-assume-safe-padding
- Specify loop trip counts
Memory performance checklist

• Use blocking to reduce L1/L2 misses
• Tune software prefetcher distance
  • Special prefetches for write only data
• Use 2MB pages when necessary
• Consider using explicit cache eviction
• #pragma vector nontemporal
  • streaming store instructions
• Padding may be necessary
Offload performance checklist

• Use asynchronous double buffering
• Avoid unnecessary data transfers
• Align data to page boundaries
• Avoid small grain offload regions
Agenda

• Introducing the Intel Xeon Phi coprocessor
  – Overview
  – Architecture

• Programming the Intel Xeon Phi coprocessor
  – Native programming
  – Offload programming
  – Using Intel MKL
  – MPI programming

• Real quick optimization list

• Summary
Little reminder...

- Up to 61 in-order cores
  - 512-bit wide vector registers
    - masking
    - scatter/gather
    - reciprocal support
    - need alignment
  - Ring interconnect
- Two pipelines
  - Dual issue with scalar instructions
  - Pipelined one-per-clock scalar throughput
- 4 hardware threads per core
- Up to 16 GB
  - ~170 GB/s
Intel® Xeon Phi™ Product Family
based on Intel® Many Integrated Core (MIC) Architecture

2013:
Intel® Xeon Phi™ Coprocessor x100 Product Family
“Knights Corner”
22 nm process
Up to 61 Cores
Up to 16GB Memory

Intel® Xeon Phi™ Coprocessor x200 Product Family
“Knights Landing”
14 nm
Processor & Coprocessor
Up to 72 cores
On Package, High-Bandwidth Memory

Future Knights:
Upcoming Gen of the Intel® MIC Architecture
In planning
Continued roadmap commitment

*Per Intel’s announced products or planning process for future products
Programming Models Summary

- The Intel Xeon Phi coprocessor supports native execution and host-centric computing with offloading

- The tool chain fully supports the traditional way of (cross-) compiling and optimization for the coprocessor

- Programmers can choose between explicit offloading and implicit offloading to best utilize the coprocessor

- MKL users can take advantage of Automatic Offload

- MPI works off-the-shelf

- You need parallelism and vectorization!
Programming Resources

- Intel® Xeon Phi™ Coprocessor Developer’s Quick Start Guide
- Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors
- Access to webinar replays and over 50 training videos
- Beginning labs for the Intel® Xeon Phi™ Coprocessor
- Programming guides, tools, case studies, labs, code samples, forums & more

http://software.intel.com/mic-developer

Using a familiar programming model and tools means that developers don’t need to start from scratch. Many programming resources are available to further accelerate time to solution.